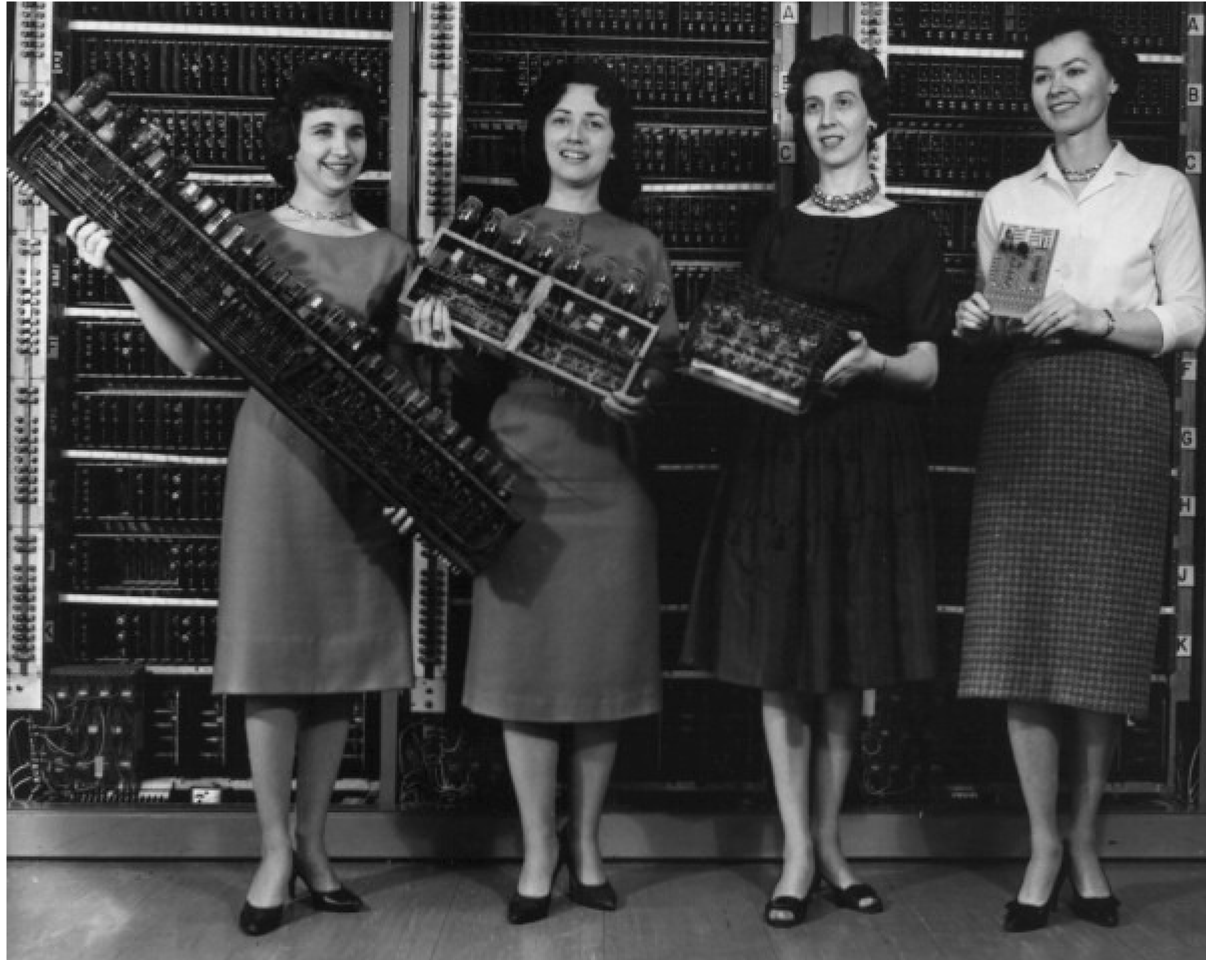
Dygresja: cztery płyty główne…



Dygresja: osobliwości C

```
/*  cos o nieistniejacym typie Boolean  */
/*  oraz o operatorze przecinkowym  */
#include <stdio.h>
#include <stdlib.h>
  int main()
```

Dygresja: osobliwości C

```
{
  int a,b,c,d,e;
  a=2; b=3; c=11;d=11,e=11;
  printf("\n c=%5d  d=%5d  e=%5d\n",c,d,e);
  c=(a<b);
  d=(a>b);
  e=((a>b),a*b,13,7); /* e=7 !! */
/* gdyby  e=(a>b),a*b,13,7;  to wtedy inna kolejność...*/
```

Dygresja: osobliwości C

```
printf("\n c=%5d  d=%5d  e=%5d\n",c,d,e);
  exit(0);
} /*   koniec   funkcji   main  */
```

oto wynik programu....

```
c=   11  d=   11  e=   11

c=    1  d=    0  e=    7
```

formatowanie kodu źródłowego

```
  program1.c
cat program1.c  |   cb     > nowy1.c
cat program1.c  |  indent  > nowy1.c
```

”Programowanie jest to dobrze zdefiniowana dziedzina działalności, o dobrze określonej strukturze, której można nauczyć się na uczelni bądź z podręczników. Uruchamiania programów to bezładny, nieukierunkowany, niewdzięczny wysiłek wykonywany pod przymusem i naciskiem, często późną nocą, z poganianiem przez niedotrzymane terminy. Nie ma ono ani metod ani reguł. Początkujący muszą polegać na własnej pomysłowości i pomocy kolegów.”

prototyp funkcji

```
#include <stdio.h>
#include <stdlib.h>


int func1(int k);
          /* tu powyżej jest deklaracja funkcji; definicja będzie poniżej */
  int main()
{
  int n;
  int k;
  k=5;
  n=func1(k);
  printf("\n n=%d\n",n);
  n=func2(k);
  printf("\n n=%d\n",n);
  exit(0);
} /* koniec main */
```

prototyp funkcji

```
int  func1(int t)
    {
      return(t*t);
    } /* koniec funkcji func1*/
int  func2(int t)
      {
            double d=10.;
            return((int)d*t);
      } /* koniec funkcji func2 */
```

funkcja - uzupełnienie

1. nie jest możliwe definiowanie funkcji wewnątrz funkcji

2. domyślnie przyjmuje się, że wszystkie funkcje dają w wyniku wartość typu **int**; funkcje dające w wyniku wartość innego typu muszą być deklarowane przed użyciem

3. jest nowy typ **void** dla funkcji, funkcja taka nie daje żadnego wyniku i użycie w niej instrukcji

   return (wyrażenie) jest błędem

4. argumenty funkcji przekazywane są przez wartość, nie przez nazwę, z wyjątkiem tablic i (niekiedy) struktur

5. funkcja nie może zmienić zawartości zmiennej przekazanej do niej jako argument

funkcja - uzupełnienie

6. wszystkie funkcje we wszystkich plikach źródłowych tworzących program i we wszystkich dołączanych bibliotekach są dla siebie jednakowo widoczne. Jedynym wyjątkiem jest funkcja zdefiniowana jako mająca klasę pamięci **static,** której zasięg jest ograniczony do pliku źródłowego, w którym jest zdefiniowana

7. reguły zasięgu deklaracji funkcji są takie same jak reguły zasięgu deklaracji zmiennych (wrócimy do tego)

8. domyślną klasą pamięci dla funkcji nie jest **static** , lecz **extern**

funkcja - uzupełnienie

Rekurencja to technika programowania polegajaca na tym, że funkcja może wywołać samą siebie (rekurencja bezpośrednia) lub wywołać inną funkcję, która z kolei wywołuje tą pierwszą (rekurencja pośrednia). Przy rekurencyjnym wołaniu funkcji są tworzone kolejne zestawy jej zmiennych automatycznych – rośnie zapotrzebowanie na pamięć, co jest głównym kosztem rekurencji. Język C udostępnia mechanizm rekurencji.

przykład: liczenie silni

rekurencyjne liczenie silni

```
#include <stdio.h>

#include <stdlib.h>

float silnia(int a);  /* prototyp funkcji   */

int   main()

{

        int k=4;

        printf("\n silnia(%d) = %f\n",k,silnia(k));

}  /* koniec funkcji main */

float silnia(int k)

{       if( k<0) {printf("\nblad!!");  exit(0); }

        if( k==0 || k==1)  return(1.0);

        return( (float)k  *  silnia(k-1) );

}  /* koniec funkcji silnia */
```

struktura programu - uzupełnienie

Każdy program musi mieć dokładnie jedną funkcję o nazwie **main**.

Zarówno zmienne, jak i funkcje muszą być zadeklarowane przed użyciem, wyjątkiem są funkcje dające w wyniku wynik całkowity.

Kolejność funkcji w pliku źródłowym nie ma znaczenia, pod warunkiem że funkcje dające wynik inny niż **int** powinny być zadeklarowane przed pierwszym użyciem.

wyrażenie

- nazwa zmiennej
- wywołanie funkcji
- nazwa tablicy
- stała
- nazwa funkcji
- odwołanie do elementu struktury
- odwołanie do elementu tablicy
- jedna z powyższych postaci z nawiasami i/lub operatorami

instrukcja - uzupełnienie

wyrażenie zakończone średnikiem

instrukcja pusta ;

if (wyrażenie) instrukcja

if (wyrażenie) instrukcja else instrukcja

while (wyrażenie) instrukcja

do instrukcja while (wyrażenie);

for(wyrażenie1; wyrażenie2; wyrażenie3)
instrukcja

instrukcja - uzupełnienie

```
switch wyrażenie_całkowitoliczbowe
        {
          case stała1: instrukcja1
          case stała2: instrukcja2
          ...
        }
```

instrukcja - uzupełnienie

```
break;

continue;

goto etykieta;

return;

return wyrażenie;
```

instrukcja **switch**

Stosuje się ją tam, gdzie należy dokonać wyboru na podstawie pojedynczego wyrażenia całkowitoliczbowego, o ograniczonym zakresie wartości

składnia *switch (wyrażenie) instrukcja*

```
int i,j,k;
switch(i)
{       case 0: k++;
        break;
        case 1: j++;
        break;
        default: j--;
        break;
}
```

instrukcja **goto**

goto *etykieta*;

....

....

etykieta:

etykieta: instrukcja

gdzie stosować: tam gdzie doprowadzi do istotnego uproszczenia zawikłanej struktury programu

instrukcja return;

instrukcja kończy wykonanie funkcji, w której występuje

```
return;
return wyrażenie;
return (wyrażenie);
```

Druga (trzecia) postać instrukcji nie może być używana
w funkcjach które zostały zdefiniowane jako mające
typ **void**

instrukcja złożona (blok)

```
{   wyrażenie_1 ; wyrażenie_2;...

}
```

/* można umieścić instrukcję złożoną wszędzie tam, gdzie składnia języka C przewiduje umieszczenie instrukcji */

instrukcja przypisania

```
        zmienna = wyrażenie ;

tesla= 1.13*4;

suma= 1.33 + f[12];

wynik= 4.56 * funkcja_jeden(5, -3);


a = b = c = d = e;
```

agregat

Jest to obiekt złożony z innych obiektów, zwanych jego elementami.

tablica, unia, struktura

l-wartość

Jest to obiekt, który może pojawić się po lewej stronie instrukcji przypisania; czyli zmienna której można przypisać wartość, co ją odróżnia od stałej.

Do l-wartości można stosować jednoargumentowy operator &, podający jej adres.

```
float x;        &x
```

wiersz kontynuacji

Znak ”backslash” \ umożliwia kontynuowanie instrukcji w następnym wierszu. Kompilator nie wymaga znaku ukośnika, z wyjątkiem przypadku ciągu znaków, a preprocesor wymaga go w przypadku makrodefinicji.

```
char a[10]=”abc\
def”
```

```
#define MAKRO(a,b,c) { func_a( ( a ), ( b ), ( c ));\
                        func_b(( a )); }
```

```
#define cube(i)  ( (i) * (i) * (i) )
```

przykład makrodefinicji

```
#include <stdio.h>

#include <stdlib.h>


#define  cube(i)  (i*i*i)

char c1='\106';

   int main()

 {

       float f;

       int n=3;

       f=-6.;

       printf(" cube %10.3f\n", cube(f) );

       printf(" cube %10d\n", cube(n) );

       exit(0);

 }      /*  koniec funkcji main  */
```

makrodefinicja

makrodefinicja jest szybką metodą zapisywania często używanych instrukcji; podaje się ja za pomocą dyrektywy #define preprocesora

```
#define nazwa_makrodefinicji(lista_argumentów)  \
        ciało_makrodefinicji
```

makrodefinicja

```
#define max(a,b)   (((a)>(b)   ?  (a)  : (b)))

          pułapki !

i =5; j = 2; i=max(++i,j);              /*   czy i = 6 ? */


i  =  ((++i)>(j)   ?  (++i)  :  (j) ;    /*   i = 7 !!       */
```

makrodefinicja – kolejność wykonywania

```
#define cube(i)  (i*i*i)
```

czy **#define cube(i) ((i)*(i)*(i))**

```
cube(j+2)

(j+2*j+2*j+2)
```

```
#include <stdio.h>

#include <stdlib.h>

#define fun(x)   ( (x) % 10  ?  1 : 0)

      int main ()

{  int k=-1;

   printf("\n %d\n",fun(5)); /* pisze 1 */

   printf("\n %d\n",fun(9)); /* pisze 1 */

   printf("\n %d\n",fun(10));/* pisze 0 */

   printf("\n %d\n",fun(11));/* pisze 1 */  exit(0);

}  /* koniec    funkcji    main   */
```

```
#include <stdio.h>
#include <stdlib.h>  /* poniżej przykład makrodefinicji */
#define fun(x,y) {double t; t=x;x=y;y=t;}
     int main ()
{ float a,b;
  a=-1;
  b=-2;
  fun(a,b);
  printf("\n %f %f\n",a,b);   exit(0);
}  /* koniec  funkcji main */
```

białe znaki

to te znaki, które w druku pojawiają się jako przestrzeń niezadrukowana – stąd ich nazwa

spacja, tabulatory poziomy i pionowy, powrót karetki,

znak nowego wiersza

\t \f \r \n

kompilator C pomija białe znaki wszędzie , oprócz ciągów znaków w cudzysłowach

typ **enum**

(jak typy int, float, double)

zmienna typu wyliczeniowego **enum** ma ograniczony zakres, który musi być podany w jej definicji

```
enum tydzien
    {poniedzialek,wtorek,sroda,
     czwartek,piatek,sobota,niedziela
    }  dzien ;
```

/ zdefiniowana zmienna typu enum tydzien o nazwie dzien */*

typ **enum**

wartość takiej zmiennej może być porównywana z WARTOŚCIAMI DOPUSZCZALNYMI

```
if(dzien==piątek) instrukcja1 else instrukcja2
```

ewentualnie przypisanie np..

```
dzien = sobota
```

również wyłącznie WARTOŚCI DOPUSZCZALNE

można przypisywać

typ **enum**

```
#include <stdio.h>
#include <stdlib.h>
  int  main()
{
      enum tydzien
      {  poniedziałek, wtorek, sroda, czwartek,
         piatek, sobota, niedziela
      } dzien1;
      enum tydzien dzien2;
      dzien1=poniedzialek;
      dzien2=wtorek;
      printf("\n dzien2=%d\n",dzien2);
}  /*   koniec funkcji  main */
```

```
#include <stdio.h>
#include <stdlib.h>
    int main ()
{    /*   iterowanie z użyciem obiektu typu enum  */
  enum tydzien{ poniedzialek, wtorek, sroda, czwartek,
                         piatek, sobota, niedziela
                       } dzien,dzien2;
   for(dzien=poniedzialek; dzien<=niedziela; ++dzien)
   { printf("dzien=%4d\n",dzien); }  ; exit(0);
}    /* koniec  funkcji main */
```

instrukcja **typedef**

Deklaracja **typedef** służy do nadania nowej nazwy istniejącemu typowi danych. Nie tworzy ona nowego typu danych.

```
typedef  typ_danych   nowa_nazwa
```

```
typedef  int   nat;

typedef  enum  tydzien  nowina;   /* poprzedni slajd  */
```

funkcja printf

```
printf(komunikat, wyrażenie1,wyrażenie2...);

printf(”dwa razy %10d wynosi %8d\n”,term, 2*term);
```

%d %f %e %g %c %s

%x %o argument będzie przekształcony do postaci ósemkowej lub szesnastkowej

%u argument będzie przekształcony do postaci dziesiętnej bez znaku

%hd argument będzie przekształcony do postaci short

funkcja scanf

```
scanf(”%d %d ”, &zmienna1, &zmienna2);
```

konwersja - wyrażenia arytmetyczne

1. argumenty **float** sa przekształcane do **double**, a **char** lub **short** do int

2. jeśli jeden argument jest **double**, drugi jest przekształcany do **double**, wynik jest typu **double**

3. jeśli jeden argument jest **long**, to drugi jest przekształcany do **long**,wynik jest typu **long**

4. jeśli jeden argument jest **unsigned**, drugi jest przekształcany do **unsigned** wynik jest **unsigned**

5. po osiągnięciu tego punktu oba argumenty muszą być typu **int**, wynik jest typu **int**

konwersja - przypisania

wartość wyrażenia z prawej strony jest przekształcana do typu lewego argumentu

(gdy zmienna int do char lub long do char, short, int to nadmiarowe najbardziej znaczące bity są tracone

gdy float do int, to gubiona jest część ułamkowa

double do float przez zaokrąglanie)

konwersja - przypisania

```
int n;
float f = -1.23;
n = f;   /* tu zajdzie konwersja,
              po konwersji będzie n = -1   */
```

Typ znakowy

```
#include <stdio.h>

#include <stdlib.h>

 int  main()

 {

        char c1='b';

        char c2='\142';

        char c3=98;

        printf(”%d   %d   %d”,c1,c2,c3);

        printf(" %c %c %c", c1,c2,c3);

        exit(1);

 }  /*  koniec funkcji  main */
```

typ całkowity int,
typ zmiennoprzecinkowy float

```
int i;

float b;
```

IEEE Standard 754 for Floating Point Numbers

http://cs.boisestate.edu/~alark/cs354/lectures/ieee754.pdf

http://www.ee.ucla.edu/~vandenbe/103/lectures/flpt.pdf

zakresy zmiennych

```
#include <stdio.h>
#include <stdlib.h>
int k=0,j;
int  main()
{
        j=1000*1000*1000;
        while(1)
        {
                k+=100;
                if(k/j*j==k) printf("\n k=%d\n",k);
                if(k<0)  { printf("\n  !! k=%d",k);  exit(0);}
        }
}  /*   koniec funkcji  main */
```

zakresy zmiennych

k=1000000000

k=2000000000

!! k=-2147483596

zakresy zmiennych

```
#include <stdio.h>
#include <stdlib.h>
float f,g;
 main()
{      f=1;
       while(1)
       {
              f*=1000;
              printf("\n   f=%f",f);
       }
}            /*   koniec funkcji  main */
```

zakresy zmiennych

f=1000.000000

f=1000000.000000

f=1000000000.000000

f=999999995904.000000

f=999999986991104.000000

f=999999984306749440.000000

f=999999949672133170000.000000

f=999999941790833820000000.000000

f=999999914697178460000000000.000000

f=999999939489602490000000000000.000000

f=999999917124474830000000000000000.000000

f=999999882462153730000000000000000000.000000

f=inf

zakresy zmiennych

```
#include <stdio.h>

#include <stdlib.h>

float f=1;

  main()

{       printf("\n");

        while(1==1)

        {       f*=0.001;

                printf("  f=%20.5e\n",f);

                if( f==0.) exit(0);

        }

}     /*   koniec funkcji  main */
```

zakresy zmiennych

```
f=    1.00000e-03
f=    1.00000e-06
f=    1.00000e-09
f=    1.00000e-12
f=    1.00000e-15
f=    1.00000e-18
f=    1.00000e-21
f=    1.00000e-24
f=    1.00000e-27
f=    1.00000e-30
f=    1.00000e-33
f=    1.00000e-36
f=    0.00000e+00
```

zasięg zmiennej, zasięg funkcji

Zasięg zmiennej jest to ta część pliku źródłowego, w której nazwa zmiennej jest widoczna dla kompilatora. Kompilator zna zmienną od tego miejsca w pliku źródłowym w którym zostaje zdefiniowana lub zadeklarowana i przestaje ją znać w końcu bloku lub funkcji, w której została zdefiniowana lub zadeklarowana, lub w końcu pliku źródłowego, w którym została zdefiniowana lub zadeklarowana – zależnie od tego co następuje wcześniej.

Zasięg deklaracji funkcji jest określony tymi samymi regułami, co zasięg zmiennej.

zasięg zmiennej, zasięg funkcji

```
int i;  int n;  extern int q;
        main()
        {
                int j;  int n;
                func();
                ...
                ...
        } /*  koniec funkcji main  */
        int k;
        func()
        {....
        ....
        }    /*   koniec funkcji func,     koniec pliku źródłowego */
```

czas życia zmiennej

Czas życia zmiennej to czas, gdy zmienna istnieje; zmienne dzielą się na dwie grupy:

1.zmienne istniejące przez cały czas działania programu; wszystkie zmienne zdefiniowane poza funkcjami oraz zmienne klasy **static** zdefiniowane wewnątrz funkcji

2. zmienne istniejące tylko podczas wykonywania funkcji, w których zostały zdefiniowane – zmienne klasy **auto**

Czas życia i zasięg zmiennej to odmienne pojęcia; zasięg jest wyznaczany w czasie kompilowania programu, a czas życia związany jest z jego wykonaniem. Np. zasięg zmiennej klasy **static** zdefiniowanej **wewnątrz funkcji** jest ograniczony do tej funkcji; jej czas życia jest równy czasowi wykonania programu

```
static   float  alfa;
```

Inicjowanie zmiennej

dla zmiennych prostych

definicja = wyrażenie; /* */

```
int k=2*5;
```

dla tablic i struktur

definicja ={lista wartości początkowych}

```
int tab[4]={1,2,3,4};
```

Definicja zmiennej - uzupełnienie

```
klasa_pamięci        typ   nazwa_zmiennej

                     float  x;

   static            int    y;

   auto              int    z;

   extern            float  a;

    register         float  b;
```

(jeszcze możliwy kwalifikator **volatile** i kwalifikator **const**)

```
    const static int alfa=11;

    volatile static int beta;
```

Klasa pamięci dla zmiennej – klasa **static**

Zmienna klasy static zdefiniowana pomiędzy funkcjami – można się do niej odwołać z dowolnego miejsca pliku źródłowego.

Zmienna klasy static zdefiniowana wewnątrz funkcji lub bloku – można się do niej odwołać tylko z wnętrza tej funkcji lub bloku.

Zmienna klasy static istnieje przez cały czas działania programu.

Klasa pamięci dla zmiennej – klasa **extern**

Zmienna klasy extern jest zdefiniowana zewnętrznie, to jest poza plikiem źródłowym lub funkcją, w których jest zadeklarowana jako extern.

Gdy zmienna jest zadeklarowana jako extern poza funkcją, to można się do niej odwołać z dowolnego miejsca pliku źródłowego. Jeśli wewnątrz funkcji lub bloku, to można się do niej odwołać tylko z wnętrza tej funkcji lub bloku.

Zmienna klasy extern nie może być inicjowana!

(Może być inicjowana tam, gdzie jest jej definicja)

Klasa pamięci dla zmiennej – klasa **auto**

Zmienna klasy auto może być definiowana wewnątrz funkcji lub bloku. Za każdym razem, gdy blok lub funkcja jest wykonywana, przydziela się jej pamięć, a po zakończeniu wykonywania pamięć jest zwalniana

Klasa pamięci dla zmiennej – klasa **register**

O ile to jest możliwe, zmienna tej klasy jest zapamiętywana w rejestrze komputera. Nie ma na to gwarancji. Zmienną klasy register można definiować tylko wewnątrz funkcji lub bloku i tam można się do niej odwoływać.

Klasa pamięci dla zmiennej – domyślna

Jeśli nie podano, to klasa pamięci jest **auto** ,

jeśli definicja jest wewnątrz bloku lub funkcji,

w przeciwnym razie jest **static**

pętla while

while (wyrażenie) instrukcja

```
while(i<k)
{
        func_jeden(++i);
        func_dwa(++j);
 }
```

pętla for

```
for(wyrażenie1;  wyrażenie2;  wyrażenie3) instrukcja
```

```
wyrażenie1;
while (wyrażenie2)
{
        instrukcja
        wyrażenie3;
}
```

```
n=0;
while (n<=7)
{
 a[n]=n;
 ++n;
}
```

pętla do-while

do instrukcja while wyrażenie

```
do
    {
        funkcja1(n++);
        funkcja2(++k);
    } while (k<j) ;
```

instrukcja **switch**

Stosuje się ją tam, gdzie należy dokonać wyboru na podstawie pojedynczego wyrażenia całkowitoliczbowego, o ograniczonym zakresie wartości

składnia *switch (wyrażenie) instrukcja*

```
int i,j,k;
switch(i)
{       case 0: k++;
        break;
        case 1: j++;
        break;
        default: j--;
        break;
}
```

Instrukcje break i continue

break;

jest używana do natychmiastowego opuszczenia pętli **while**, **for**, **do...while** lub instrukcji **switch**

continue;

służy do wcześniejszego zakończenia kolejnej iteracji pętli **for**, **do**...**while** lub **while**; wykonanie pętli jest kontynuowane

co ma zmienna?

•nazwę (identyfikator)

•typ (int float ...)

•klasę pamięci (extern, static, auto, register)

•zasięg (w której części programu jest dostępna)

•czas życia

(dokładność reprezentacji wynika z typu)

znak końca linii: UNIX i WINDOWS

```
#include <stdio.h>
int main( )
{
  int c1;
/*  char c1 ??! */
  while(1)
  {
    if((c1 = getc(stdin)) == EOF)  break;
    if(c1==13) continue;
    putc(c1,stdout);
  }
}  /*  koniec funkcji  main */
```

Częste błędy w C - 0

kolejność obliczeń; język C nie gwarantuje, że argumenty w wyrażeniu będą obliczane w jakiejś ustalonej kolejności.

i=0;

func(++i,i+2);

może to być równie dobrze **func(1,3)**

jak i **func(1,2)**

Częste błędy w C - 1

wyrażenie   if(i = 9) jest poprawne składniowo

prawdopodobnie jest to błąd – jest zawsze

prawdziwe

Częste błędy w C - 2

definiowanie tablicy n-elementowej i sięganie po element numer n

```
float rak[10];

rak[10] = 134;  /*   błąd !! */
```

Częste błędy w C - 3

niepoprawne adresowanie pośrednie

(gdy będziemy mówić o wskaźnikach)

zapomina się o napisaniu * lub pisze się o jedną gwiazdkę za dużo

Częste błędy w C - 4

definiowanie wskaźnika bez przydzielania pamięci dla obiektu przez niego wskazywanego i następnie użycie go

Częste błędy w C - 5

pisanie o jedną parę nawiasów za mało

```
if(i = max(a,b) == b);
```

i otrzyma wartość jeden lub zero

```
if( (i = max(a,b)) == b)
```

i otrzyma wartość większą z a,b

”nadmiarowa para nawiasów nie szkodzi, a brakująca para nawiasów nie pomaga”

Częste błędy w C - 6

błąd w wierszu wychodzącym poza ekran; należy pisać rozsądnej długości linie w programie źródłowym

Częste błędy w C - 7

błędnie zakończone komentarze

```
/*                                        * /
                      /* tu powyżej niepotrzebna spacja!! */
```

Częste błędy w C - 8

zmiana w funkcji dostarczanej przez kogoś innego (np. w funkcji bibliotecznej)

Częste błędy w C – 9
użycie złej wersji pliku

poprawiona wersja pliku źródłowego nie zostaje zapamiętana; potem wykonuje się kompilację i próbuje uruchomić program posługując się dawną wersją pliku i otrzymuje się te same błędy;

Częste błędy w C – 9
użycie złej wersji pliku

nie zauważono, że poprawiona wersja nie dała się skompilować i ponownie zostaje uruchomiona stara wersja

Częste błędy w C – 9
użycie złej wersji pliku

kompilujemy nową wersję, ale konsolidujemy (linker) starą wersję

Częste błędy w C – 9
użycie złej wersji pliku

uruchamiamy program z dawnym zestawem danych zamiast z nowym

Częste błędy w C – 9
użycie złej wersji pliku

wydruk programu, na którym pracujemy, nie odpowiada aktualnemu programowi źródłowemu

Częste błędy w C – 9
użycie złej wersji pliku

różne kombinacje powyższych błędów

Częste błędy w C - 10

błędne przyjęcie, żę plik danych zawiera dokładnie to, czego się spodziewamy

(przy uruchamianiu programów wszelkie założenia a'priori są podejrzane; ważne są tylko fakty)

Częste błędy w C - 11

niezgodności między argumentami formalnymi a aktualnymi; kompilator C nie sprawdza, czy argumenty formalne i aktualne odpowiadają sobie zarówno co do typu jak co do liczby

w systemie UNIX można użyć programu lint do zlokalizowania takich błędów; program ten często wskazuje zbyt dużą liczbę błędów

Częste błędy w C - 12

opuszczenie nazwy funkcji w wywołaniu

```
printf(”i = %d”, i);        oraz        (”i = %d”, i);
```

Ta druga instrukcja jest w pełni poprawna, natomiast nic nie robi.

Częste błędy w C - 13

Zamiana stałej nie we wszystkich miejscach

Jak unikać tego błędu:

```
#define  ROZMIAR 20 /* to rozmiar macierzy */
int  macierz[ROZMIAR];
...
....
  for( k=0;   k<ROZMIAR;  ++k) ....
...
```

Częste błędy w C - 14

zapomnienie przecinka w wywołaniu funkcji

Dlaczego?   * & -   mają różne znaczenie w zależności od tego czy są operatorami dwuargumentowymi czy jednoargumentowymi

(mnożenie, iloczyn logiczny bitowy, odejmowanie)

(wskazanie pośrednie, adres obiektu, negacja wartości)

```
func(i, &k)                func(i &k)
```

tablice, tablice wielowymiarowe

```
float tab1[12];

int tab2[12][10];

char tab[20];
```

łańcuchy znakowe

W języku C, nie ma osobnego typu ”łańcuch znakowy”, łańcuchy znakowe są budowane przy użyciu tablic znaków. Znak specjalny ’\0’ jest używany do zaznaczenia końca łańcucha.

```
char par[20]=”Kolec”;

char rak[]=”Rak”;

        printf(”\n  %s  %s\n”,par, rak);

        scanf(”%s”,par);    /*  nie ma operatora & */
```

łańcuchy znakowe (rozdział 5 w S.O.”J. C”)

kilka funkcji przetwarzających łancuchy

strcpy(lan1, lan2)	kopiuje lan2 do lan1
strcat(lan1,lan2)	dokleja lan2 na końcu lan1
strlen(lan1)	podaje długość lan1
strcmp(lan1,lan2)	zwraca zero jeśli lan1 jest równy lan2,

fgets(nazwa, sizeof(nazwa), stdin)

(nazwa to zmienna łańcuchowa, np. tablica znakowa)

przenośność programu

Język C nie gwarantuje przenośności programowania, ale definicja języka nie jest zależna od konkretnego komputera, więc... . Obydwa kompilatory muszą być standardowe lub rozsądnie bliskie przyjętym standardom. W programie nie należy czynić założeń dotyczących własności sprzętu komputerowego; te własności to ustawienia obiektów, rozmiary danych różnych typów, kolejność zapamiętywania bajtów (np. że bajty mniej znaczące mają adres mniejszy niż bajty bardziej znaczące).Kompilatory mogą dopuszczać różne maksymalne długości nazw. Niestaranność przy przekazywaniu argumentów może przejść niezauważona na jednym komputerze, ale spowodować kłopoty na drugim.

struktura

deklaracja struktury *oznacznik* jest równoważna deklarowaniu nowego typu

struct *oznacznik* {lista_deklaracji};

/* deklaracja struktury nazwanej *oznacznik* */

struct {lista_deklaracji} nazwa_zmiennej;

/* deklaracja nienazwanej struktury i zdefiniowanie zmiennej */

struct *oznacznik* {lista_deklaracji} nazwa_zmiennej;

/* deklaracja struktury i definicja zmiennej o typie struct *oznacznik* */

struktura

```
struct  complex {        double real;
                         double imaginary;
                }        alfa;


struct complex beta[3], gamma;
```

unia

Unia to zmienna, która służy do pamiętania kilku obiektów różnych typów w tym samym obszarze pamięci. Składnia deklaracji unii jest praktycznie taka sama jak składnia deklaracji struktury.

```
union  alfa{        int i;
                    char t;
               } bak[10];
union  alfa  mak;
```